

## Redesigning a Graphics Test Program Using C++

Jerry Fitzpatrick

Are you considering a C++ rewrite of an existing C program? Are you skeptical about the benefits on OO design might provide? My experience rewriting an MS-DOS program for testing graphics adapter cards on IBM-PC compatible computers is related here. Besides describing the “lessons learned”, I’ll also provide some valuable guidelines you can use to successfully reengineer existing systems using OO and C++.

Prior to my redesign, I had converted the original C program to compile as a C++ program. Several problems cropped up, as described in my previous article.<sup>1</sup> However, this conversion allowed me to add functionality using C++ classes even though the overall design was not object-oriented. In this article I’ll show that a redesign can improve modularity, reduce code size, and produce a significant number of reusable components.

Modern graphics adapters are very complex and provide a staggering range of features and operating modes. Although I intend to focus on C++ and the redesign process I first need to explain some of the basic functionality of graphics adapter hardware.

### Anatomy of a Graphics Adapter

The major components of a graphics adapter are the CRT controller (CRTC), the digital-to-analog converter (DAC), and video memory (VRAM). The adapter can be designed to interface with the ISA bus architecture, but higher performance is possible using the PCI or VESA local bus. The relationship between the adapter components is shown in Figure 1.

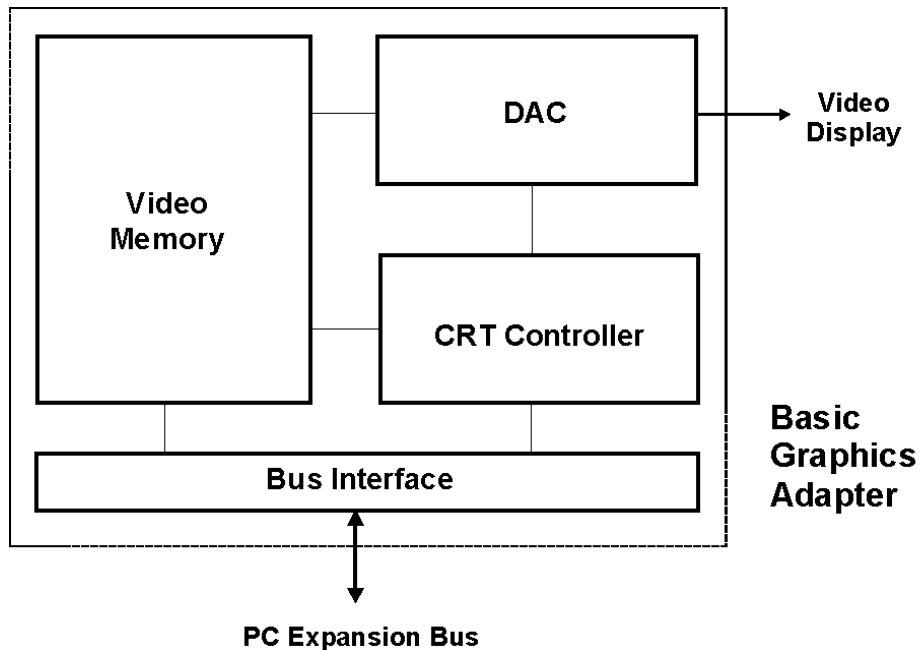


Figure 1 – The relationship between the adapter hardware components.

The basic CRT controller functions are based on the Motorola 6845, the mother of modern graphics controllers. These functions control the scanning of the electron beam inside the CRT, which determines the width and height of the display. The CRTC is also responsible for displaying the text cursor. Besides CRT operations, many other functions are needed for complete graphics control.

Advanced graphics controllers have built-in “accelerators” that speed up a variety of drawing operations. Accelerators often include line drawing engines, polygon or trapezoid drawing engines, and a “bit blitter”. The blitter is used to copy a rectangular section of display memory to another part of the display or to system memory. This invaluable to graphical operating systems like Microsoft Windows where icons and windows are continually being repositioned and redrawn.

In many ways the DAC is the heart of the graphics adapter. Synchronized with the CRTC, it constantly scans each byte of video memory converting the digital information to three analog color signals for the monitor to display. The performance of a graphics adapter is generally limited by its DAC, which must operate at incredibly high speeds. For example, a typical 1024 by 768 pixel display refreshed at 60 times per second requires the DAC to paint almost 50 million pixels per second!

The DAC can operate in several different modes depending upon how the video memory data is to be interpreted to produce color. In direct color modes five, six, or eight bits per color are directly converted to an equivalent analog output. In pseudo-color mode an eight-bit code is converted to a 24-bit RGB color value using a RAM lookup table built into the DAC. This lookup table (or “color palette”) can be modified by the application to allow any 256 of 16 million distinct colors to be chosen. Figure 2 shows how the DAC uses color palette information.

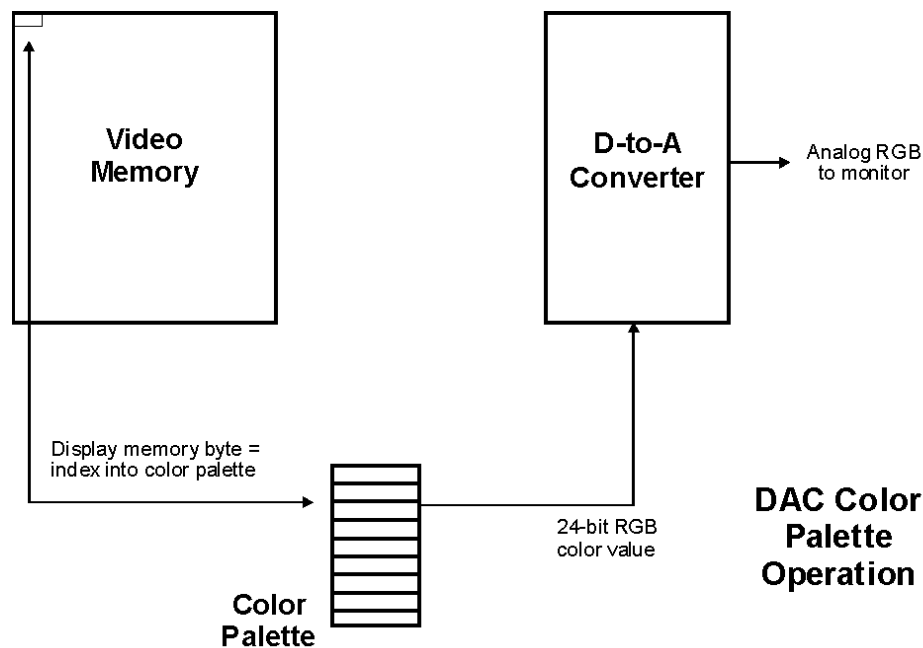


Figure 2 – DAC using color palette information

Video memory contains the digital equivalent of images displayed on the monitor. The amount of video memory required for a display depends upon the screen size and color mode. A 640 by 480 pixel pseudo-color display requires only 307,200 bytes. On the other hand, a 1024 by 768 true-color (24-bit) display requires 2,359,296 bytes. The main reason that high-resolution adapters are expensive is the need for megabytes of video RAM.

PC-based graphics adapters generally provide access to video memory in either of two ways. In linear mode, video RAM is mapped to system memory space where it can be directly addressed. This mode is the most efficient and easiest to program, but cannot be used by MS-DOS programs because pointer addresses are limited to 20 bits. For this reason, adapters can also be programmed to allow video memory paging in which 64K or smaller “pages” of RAM are mapped to a fixed memory address (usually A000:0000).

This high-level description just scratches the surface of graphics adapter technology. For details about graphics hardware and programming, refer to books by Richard Ferraro<sup>2</sup> and Michael Abrash<sup>3</sup>.

### Initial Considerations

The original diagnostic program I converted was written in C. It provided tests for several different adapters. When the program started up it identified the adapter electronically and set a global variable with the corresponding id code.

In classic C programming style, the program made liberal use of ‘if/else’ and ‘switch’ statements to perform different operations depending upon the id code. This technique was manageable for a few different adapters, but quickly became a maintenance headache as more were added.

All CRT controllers and digital-to-analog converters operate on similar principles. However, register addresses and bit positions are usually different. Even so, the core CRTC, color palette, and BIOS operations are the same from adapter to adapter.

For program design it’s very important to distinguish between the physical and functional (logical) relationships of components. Some graphics controllers, for example, use an integrated DAC while others use an external DAC. In both cases the DAC is functionally connected to the CRTC in the same way. Nevertheless, the DAC often contains additional features that differ from manufacturer to manufacturer.

I’ve found that the reusability of a C++ class is proportional to the accuracy and robustness of the abstraction. The abstraction, in turn, is usually related to the functional – not physical – aspects of the entity being modeled. I don’t want to over-emphasize the functionality because a classic functional decomposition is inappropriate for an OO design. Nevertheless, examination of the functionality can often lead you to identify classes more quickly for applications like this one. This strategy generally becomes less viable for the lower layers of the program, however, because these classes become increasingly abstract.

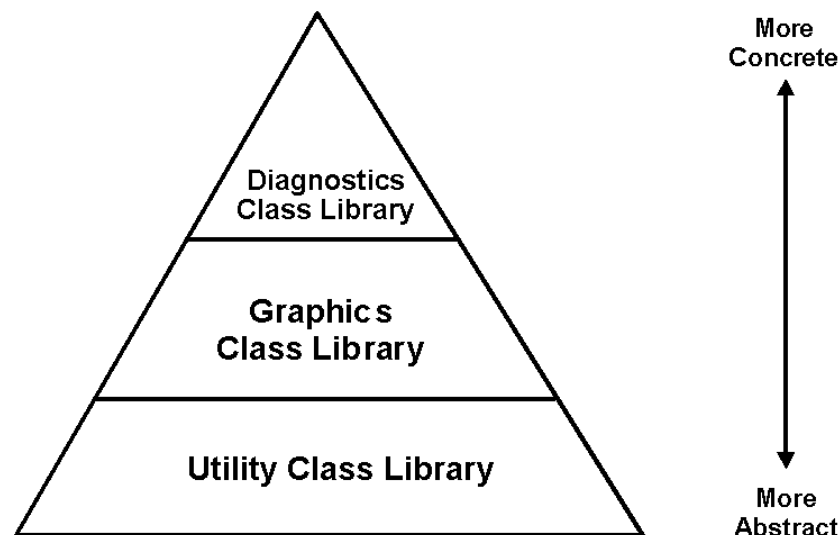


Figure 3 – Class library hierarchy

My key point is to look past the physical aspects of the problem (in this case, the physical hardware) to see the inner functional aspects. For instance, a class combining the functions of the CRTC and DAC (because they’re integrated in some chips) would be less robust than a separate class for each one. By separating the CRTC and DAC functionality into two classes you increase the flexibility of the design by being able to “mix and match” as needed.

In addition to the OO architecture, I wanted to provide the code in the form of reusable libraries. From experience, I've found that a layered approach leads to more stable, maintainable programs. I reasoned that the hierarchical design shown in Figure 3 would provide both stability and a high degree of code reuse. This is because reuse stems from abstract components, and these components dominate the lower levels of the hierarchy. This approach, of course, echoes the spirit of the standard runtime library and foundation classes shipped with compilers.

For this application the base of the hierarchy consists of generic utility classes for register access, keyboard I/O, and so forth. On top of this is a collection of generic graphics classes for handling graphics mode setting, color palettes, and screen displays. In theory these classes could be reused in non-testing applications ranging from utility programs to games. At the very top of the pyramid are classes specific to the test application. These include testing algorithms and user interface controls.

Even with a well-planned design strategy one of the obstacles to creating a reusable design is imperfect knowledge about the application.

To create a good class abstraction for an item, it helps to understand the details about the category of items that will be modeled with it. For example, creating a good video DAC class requires detailed knowledge about the DACs supplied by a variety of manufacturers. The more you know about the qualities of these specific DACs, the better the abstract DAC can be.

Since good abstractions are formed by a study of specific cases, let me illustrate the point by examining a hypothetical, but viable, product called the Meson graphics adapter. Let's assume that the design uses a Matrox Titan graphics controller, Texas Instruments TVP3026 DAC, and four megabytes of video memory. For convenience, we'll look at three phases of the design process: examining specific cases, creating a reusable architecture, and refining the initial design.

### Phase 1 – Examining Specific Cases

The Titan graphics controller contains circuitry for the CRTIC, a PCI bus interface, line drawing, trapezoid drawing, and bit-blitting. An external DAC provides the color palette and hardware cursor functions in addition to its digital-to-analog duties. Video RAM is accessible to the DAC and the system bus, although one register in the Titan controller must be used to access the RAM in paged mode.

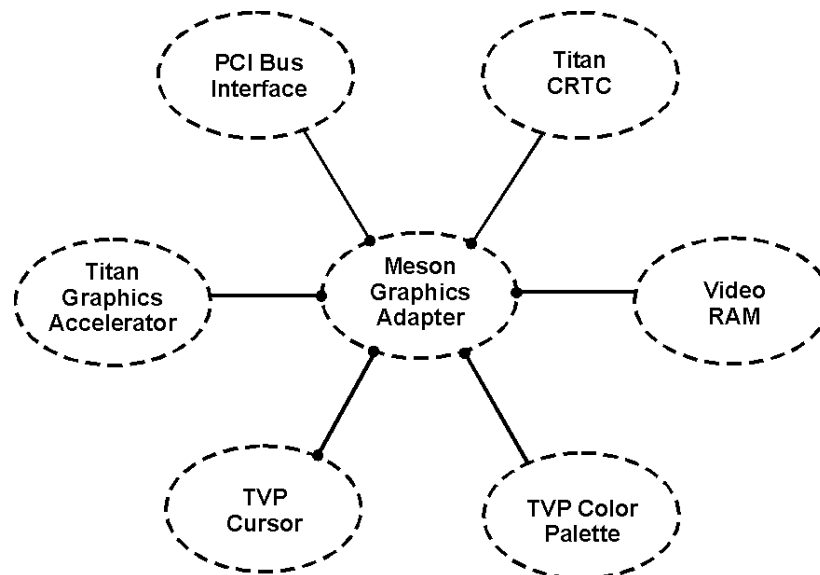


Figure 4 – Meson Graphics Adapter Design

Figure 4 shows how an object-oriented architecture could be applied to this adapter. The DAC has been split into the separate functions of hardware cursor and color palette (the digital-to-analog

conversion is essentially invisible to the program). Likewise, the Titan's functionality has been split into the CRT controller, PCI interface, and accelerator. The only purpose of video RAM is to store display data.

Note that these classes are contained in the graphics adapter class. In this regard, the adapter class becomes the equivalent of the hardware printed circuit board with its copper traces. This type of mapping between the hardware and software often makes the architecture easier to understand. It is not always appropriate to apply this reasoning at all levels of the software however. For instance, there may not be a hardware equivalent, or a different model may be more efficient or easier to understand.

Incremental development is often used for object-oriented software. This approach is very different from the classic waterfall model that separates design and coding into distinct phases. Instead, incremental development unites the design and coding phases to provide iterative refinement of the program.

On the surface, this may sound as if analysis and design can be glossed over prior to coding. This is not the intent of the approach. The incremental development model is simply an admission that designs are rarely perfect from the start.

Instead of recommending early coding, the goal of incremental development is to verify the design frequently during coding and make adjustments as necessary. My personal version of this process is to perform a top-down design followed by a bottom-up implementation. I think of the initial design as a hypothesis that gets tested during coding. This gives me a good way to verify or correct elements of the design during program construction. In this regard the use of well-tested, reusable class libraries means I don't always have to start implementing at the very bottom.

I must confess that my knowledge of graphics adapter varieties was somewhat limited at the beginning of this project. I had learned about some of the differences by studying the existing test program, but analyzing old sparsely-commented code is an arduous task.

Although many computer graphics books explain general graphics concepts, most do not describe specific adapters or controllers (a notable exception is the Ferraro book cited earlier). Ultimately I had no choice but to refine my OO architecture during the implementation.

My understanding of bit-blitters was the weakest (you'll recall that bit-blitters copy portions of the screen at high speeds). The programming aspects of bit-blitters also happen to vary widely from manufacturer to manufacturer. I did some research and took a stab at a class design, but it changed at least four or five times before coming truly reusable. It is interesting to note that bit-blitters are so complex that very few manufacturers have even worked out all their hardware bugs!

## Phase 2 – Creating a Reusable Architecture

The design shown in Figure 4 works well for the Meson adapter, but how can it be modified to work for any adapter? We know that there are major differences between graphics chips, especially with respect to the hardware cursor and graphics accelerator. In fact, the extent of the differences in future adapters is hard to predict, even though we will probably perform many of the same tests for each new chip. For these reasons, I decided to put polymorphism to good use by designing an inheritance hierarchy.

Inheritance is a very good way to connect items whose purpose or behavior is similar but whose operation or mechanism is quite different. An abstract base class provides for a uniform interface, while subclasses take care of the implementation details.

To this end, we can envision a base bus interface class that can be customized through inheritance to become an ISA, VL, EISA, or PCI bus interface class. Likewise, we can envision base CRTIC, graphics accelerator, and hardware cursor classes. Because color palette and video memory operation is the same for most adapters, these classes do not typically need customization.

The graphics adapter class embodies the test algorithms and user interface functions. Most tests are identical from adapter to adapter, but others may need modification for specific adapters. Additionally, some may need new tests that are hard to foresee. Again, these are grounds for the use of inheritance.

Figure 5 shows the revised Meson architecture based upon reusable, mostly abstract, base classes.

You may be wondering whether there is too much abstraction and layering in this design. After all, the revised design is considerably more complex in terms of the number of different classes. Don't we want to avoid complexity?

Well, yes, complexity is one of the main enemies of maintainable software. In fact, if this were the only graphics adapter we'd ever want to test the design would be overkill. My goal, however, was to provide a reusable, extensible software architecture for testing any foreseeable graphics adapter. This is no small task!

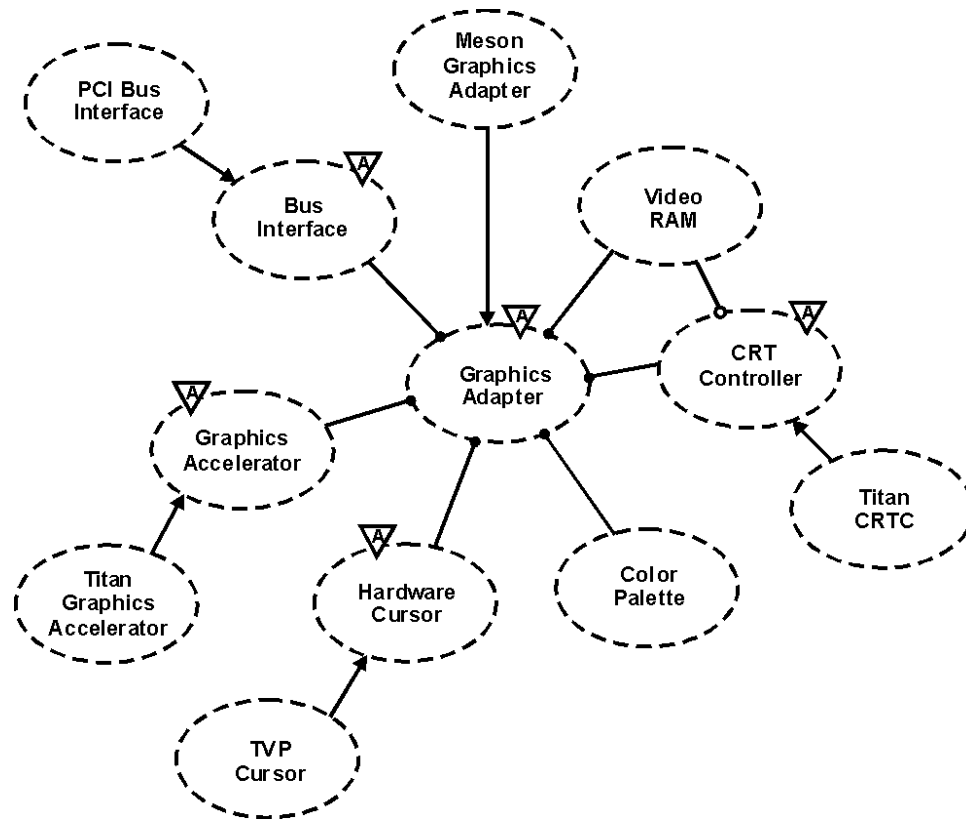


Figure 5 – Improved Meson Adapter Design

Complexity tends to be multi-dimensional and difficult to quantify. An open, extensible design is almost always more complex than a closed design because additional abstractions must be put in place to support the extensibility. This creates a steeper learning curve, but generally reduces the long-term effort required to add features and maintain the program. In this sense, the overall complexity could be considered reduced.

The graphics adapter architecture – an abstract base class containing several other abstract base classes – is quite common in object-oriented designs. This is because the definition of these classes and relationships between them form a framework or design “pattern”.

This framework is important because – like it or not – any programmer developing a new adapter is stuck with it. A good framework should lead to quick, reliable implementations. A bad framework is guaranteed to produce complaints or even a redesign! Robert Martin gives some relevant design examples in his recent book (see Martin<sup>4</sup>).

The architecture shown in Figure 5 provides considerable reuse because most of the functionality for each object is implemented in its base class. Although each of the objects can be customized, only the adapter class is truly extensible. One reason for this is the way that inheritance works in C++.

For instance, let's say that another programmer wants to add a method called Mode3D to his customized CRT controller subclass. Since most graphics chips don't have a 3D mode of operation, Mode3D is not a method of the base CRT controller class. The programmer can add the method to his class and use it privately, but the rest of the application has no way to use it because it hasn't been defined in the base class.

The open-closed principle (see Meyer<sup>5</sup>) advocates that designs be closed for implementation yet open for extension. Modifying a base class to add a (pure virtual) method does not satisfy these criteria, and creates problems for other users of the framework. Does this mean that the design is flawed?

Well, to begin with, every design has constraints. A framework intentionally provides boundaries within which programmers are expected to work. Although the framework limits a programmer's choices, he benefits by being able to reuse existing base class functionality.

Although the design of the test program could be made more extensible, I believe the additional complexity would outweigh the benefits. This is especially true since the adapter class can be extended to handle most special cases.

### Phase 3 – Refining the Initial Design

Note that the VideoRam and ColorPalette classes are not abstract in the design shown in Figure 5.

For most hardware designs, the color palette registers are at standard I/O locations and operate identically. For this reason the ColorPalette class is sufficient by itself and there is no need to specialize it using inheritance.

I had wanted the VideoRam class to be concrete since RAM is the same for every design... well, almost. Remember that special CRT controller register needed to set the RAM memory page? Its address and usage varies considerably from controller to controller. Does this mean we should also specialize VideoRam using inheritance?

Although it's often a good choice, inheritance is not the only tool with which to handle special cases. Class templates are perhaps the most elegant form of specialization, although they can be hard to manage in large systems due to compiler quirks.

I chose a time-honored technique, parameterizing the arguments to methods to solve my VideoRam problem. Although this may seem low-tech by comparison, it can be the most effective way to handle special cases. Specifically, the constructor of the VideoRam class is:

```
VideoRam(char *baseAddress, ulong size, CrtCtrl &vidctrl);
```

This constructor allows the base address and size of video RAM to be set dynamically. The reference to the CrtCtrl class provides access to the public member function:

```
ulong SetPageOffset(char *address);
```

This function takes an absolute memory address as a parameter, sets the correct memory page corresponding to that address, and returns the byte offset into the memory page. By using this function, the VideoRam class looks exactly like a contiguous region of memory even when it's operating in paged mode. That's a very useful abstraction! The only down-side is that the CRT controller object must be constructed before the VideoRam object.

Many of the graphics tests involve setting a specific video mode, then drawing a test pattern on the screen and having an operator inspect the display for defects. Creating test patterns using text modes is very different than creating them using graphics modes.

For simplicity I chose to draw text displays using the video BIOS. Unfortunately, the BIOS is too slow for graphics displays. To further complicate matters, pixel painting in graphics modes works differently depending upon the color depth and color plane organization. My first thought was: "What classes should the drawing routines belong to?"

At first I thought it should be the adapter class. Later it seemed like the CRT controller class would be a better fit. Day after day the number of drawing methods grew until the CRT controller class started bulging like a hippo in Spandex. This was my cue to step back, reevaluate my true drawing requirements, and modify the design on that basis.

After another day or two I had a solution. I developed a `TextDisplay` class for drawing test displays and a `GraphicsDisplay` class for drawing pixel-oriented displays. These classes could be instantiated as needed to draw a test pattern. Better yet, by passing a few parameters into the constructor, they could be used in any video mode with any graphics hardware! This is a good example of how OO designs can evolve to use classes having no direct relationship to any physical object.

In some well-traveled domains it's common for designs to use objects familiar in the domain. For instance, accounting and business data-processing applications often model invoices, ledgers, or customer records as objects. This is a good starting point for OO designs because of its naturalness. However, the same reasoning is difficult to apply in more abstract domains (such as systems software) because there is usually no "real world" equivalent. Under these circumstances it's perfectly okay to design unfamiliar objects provided they're not arcane or bizarre.

### Implementation

Naturally there were many more design decisions than I've described here. As with the drawing routines, I found and corrected several other design problems during coding. Although there remain a few minor rough spots, I am quite pleased with the results.

Table 1 shows the contrast between the original test program and its object-oriented equivalent.

<b>Table 1 – Comparison of Original and Redesigned Application</b>		
	<b>Original Program</b>	<b>Redesigned Program</b>
<b>C Language (C++ Language)</b>		
Number of source files	64	27
Number of header files (Note 1)	72	37
Source lines of code (Note 2)	27,676	7,120
LOC per file (mean)	432	264
LOC per file (standard deviation)	330	317
<b>Assembly Language</b>		
Number of source files	15	4
Number of header files	6	0
Source lines of code (Note 2)	6,675	1,189
LOC per file (mean)	445	297
LOC per file (standard deviation)	243	86
Note 1 - not including standard library headers		
Note 2 – not including header file LOC		

This is somewhat of an "apples and oranges" comparison. The statistics for the redesigned program reflect only the base implementation; not any specific adapter code. The original program, on the other hand, contained implementations for roughly six different adapters. Even so, the redesigned program provides additional capabilities such as drawing primitives for all graphics modes, whereas the old program handled only pseudo-color modes.

So, how much additional code is needed to create a specific adapter? Having developed several adapters based on this framework, the answer seems to be roughly 1,000 lines of code, depending upon the extent of adapter differences. This equates to about 15% of specialized code, or 85% reuse!

These figures imply that the redesigned program is less than half the size of the original. This should not be too surprising since most programs get fatter over time. This is because programmers typically add new features without taking full advantage of the existing components.

You'll not that the amount of assembly code was considerably reduced, although not eliminated. Most of the assembly code is used to provide the fastest possible testing of video RAM. I had initially used C++ routines, but was able to decrease testing time by about 300% by using high-optimized 32-bit assembler.

It's possible to interface assembly routines directly as C++ class member functions, but the interface is compiler-dependent. By contrast the C routine interface is very standardized and, for this reason, I chose to interface the assembler routines this way.

Before leaving the subject of implementation, I'd like to make a specific recommendation. Bertrand Meyer, author of the Eiffel programming language, makes a clear and compelling argument for the use of "programming by contract".

The basic idea is that every routine in a program has duties to perform given constraints specified by preconditions and postconditions. For example, a routine that calculates the square root of a number requires that the number be non-negative.

This is a valuable and powerful technique for writing more reliable programs. The Eiffel language uses the special keywords "require" and "ensure" for checking these conditions, whereas C++ does not have this capability.

Some programmers use the "assert" macro to simulate the feature. In MS-DOS an assert causes program termination, which I find annoying. Therefore I've adopted a gentler approach by combining C++ macros and output streams.

To begin with, I build my program with a module that constructs a global error logger like this:

```
#ifdef DEBUG
#include <fstream.h>
ofstream errlog("err.dat");
#endif
```

In a separate header file, I declare the error logger and define my alternative to the assert macro:

```
#ifdef DEBUG
#include <fstream.h>
extern ofstream errlog;
#define ENSURE(x) \
    if (!(x)) errlog << "Ensure failed at " \
    << __FILE__ << " line " << __LINE__ << endl;
#else
#define ENSURE(x)
#endif
```

I then use the ENSURE macro to state my preconditions and postconditions which, if violated, are reported in the error logger file. For example:

```
int SearchFileForName(const char *name)
{
    ENSURE(name != 0);
    ...
}
```

will add a message to the error log file if, for any reason, the pointer to "name" is zero (an obviously invalid condition).

An important feature of this technique is that checking is compiled out whenever the symbol DEBUG is undefined. In this way the overhead of the validation code is cleanly removed once the program is working correctly.

## Conclusion

A good object-oriented design can allow significant code reuse and extensibility to custom applications. To create a reusable framework, however, you should:

1. Study the similarities and differences between specific instances of the thing being modeled, paying special attention to the differences.
2. Create base classes that define the properties and behavior of an idealized version of the things being modeled.
3. Use inheritance to define and implement the differences between the ideal and actual thing.
4. Refine your design by noting implementation problems and making appropriate design changes.

As we've seen, redesigning an application using object-oriented C++ can reduce code size and provide substantial reuse of components. It can also ease the frustration and cost of maintenance.

Every design and designer is unique. By following these guidelines, though, you should be well on your way to achieving viable object-oriented architectures.

## References

1. Fitzpatrick, J. "Case Study: Converting C Programs to C++" *C++ Report*, Feb. 1996.
2. Ferraro, R.G. *Programmer's Guide to the EGA, VGA, and Super VGA Cards*, third edition, Addison-Wesley 1994.
3. Abrash, M. *Zen of Graphics Programming*. The Coriolis Group 1995.
4. Martin, R.C. *Designing Object-Oriented C++ Applications Using the Booch Method*. Prentice-Hall 1995.
5. Meyer, B. *Object-Oriented Software Construction*. Prentice-Hall 1988.